Luthier: Dynamic Binary Instrumentation Framework Targeting AMD GPUs

Matin Raayai-Ardakani, David Kaeli, Norman Rubin

Northeastern University Computer Architecture Lab (NUCAR)

Boston MA, USA

Motivation

Binary Instrumentation

- A technique of inserting code into a binary to observe its dynamic behavior
  - **Dynamic binary instrumentation (DBI)** injects code at runtime
  - **Static binary instrumentation (SBI)** injects code when the binary is not running
- Commonly used for:
  - Profiling
  - Hybrid simulation
  - Fault injection
- Notable CPU instrumentation frameworks:
  - DynInst
  - Intel PIN
- Notable GPU instrumentation frameworks:
  - NVBit
  - GTPin

Observation

- Previously, no DBI framework was available for AMD GPUs
  - Despite openly available docs for the Instruction Set Architecture (ISA)
  - Despite the software stack being open source
- Efforts for AMD GPU instrumentation at the time:
  - DynInst team was in early stages of adding AMD GPU support
  - AMD Research/Darche et. al statically instrumented at the LLVM IR level using compiler plugins
- Prior state-of-the-art in GPU DBI is closed source
  - Lack of transparency in the instrumentation pipeline hinders academic research

Goals

Primary Goal: Create an open-source DBI framework for AMD GPUs

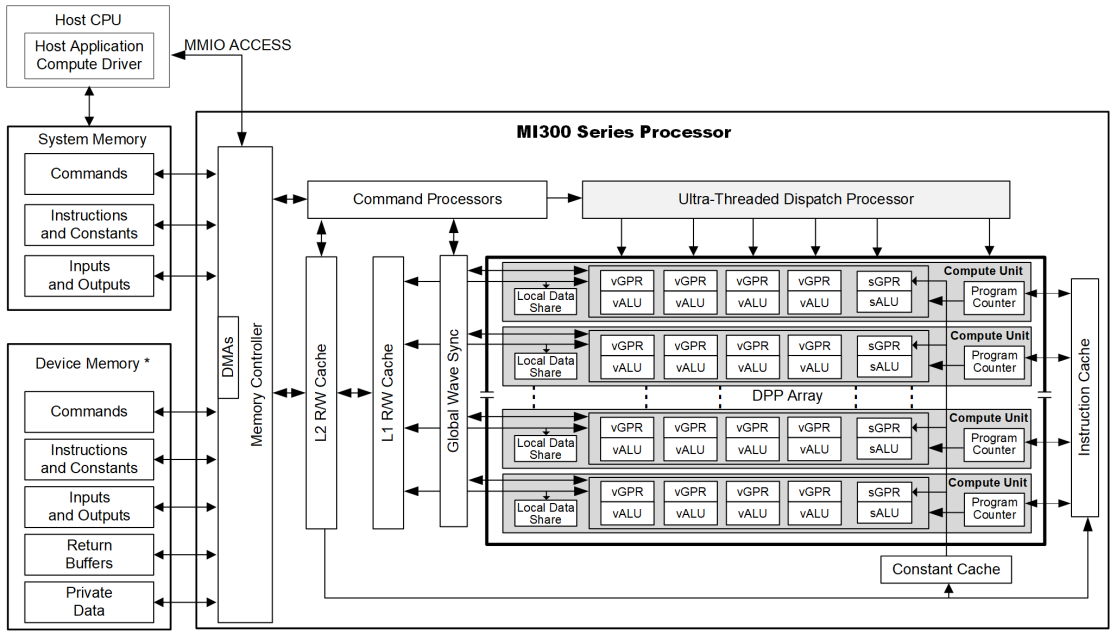
- DBI for ultimate flexibility
  - No need to access the source code
  - No need for re-compilation of the entire application (e.g., vendor libraries)
  - Selective instrumentation for reduced overhead
- Don't re-invent the wheel
  - Leverage NVBit and GTPin designs as much as possible
  - Improve design aspects when opportunities arise
- Work seamlessly with AMD's ROCm stack
  - Don't rely on modifications not approved by the vendor

Result: Luthier, our AMD GPU DBI framework

Background

Notable AMD CDNA GPU Features

- **Registers:**
  - 32-bit Scalar General-Purpose Registers (SGPR)
    - Shared among the threads in a wavefront
    - Primarily used for control flow
    - Operated on using Scalar instructions
  - 32-bit Vector General-Purpose Registers (VGPR)
    - Unique to each work-item
    - Operated on using Vector instructions
- **Special SGPR Registers:**
  - EXECute (`EXEC`) mask SGPR pair
    - Indicates if a thread ignores or executes vector operations
    - Does not apply to scalar operations
  - Flat Scratch (`FS`) SGPR pair
    - Holds the flat address of the wavefront's scratch base
- The kernel must specify the number of registers and amount of LDS memory used in execution

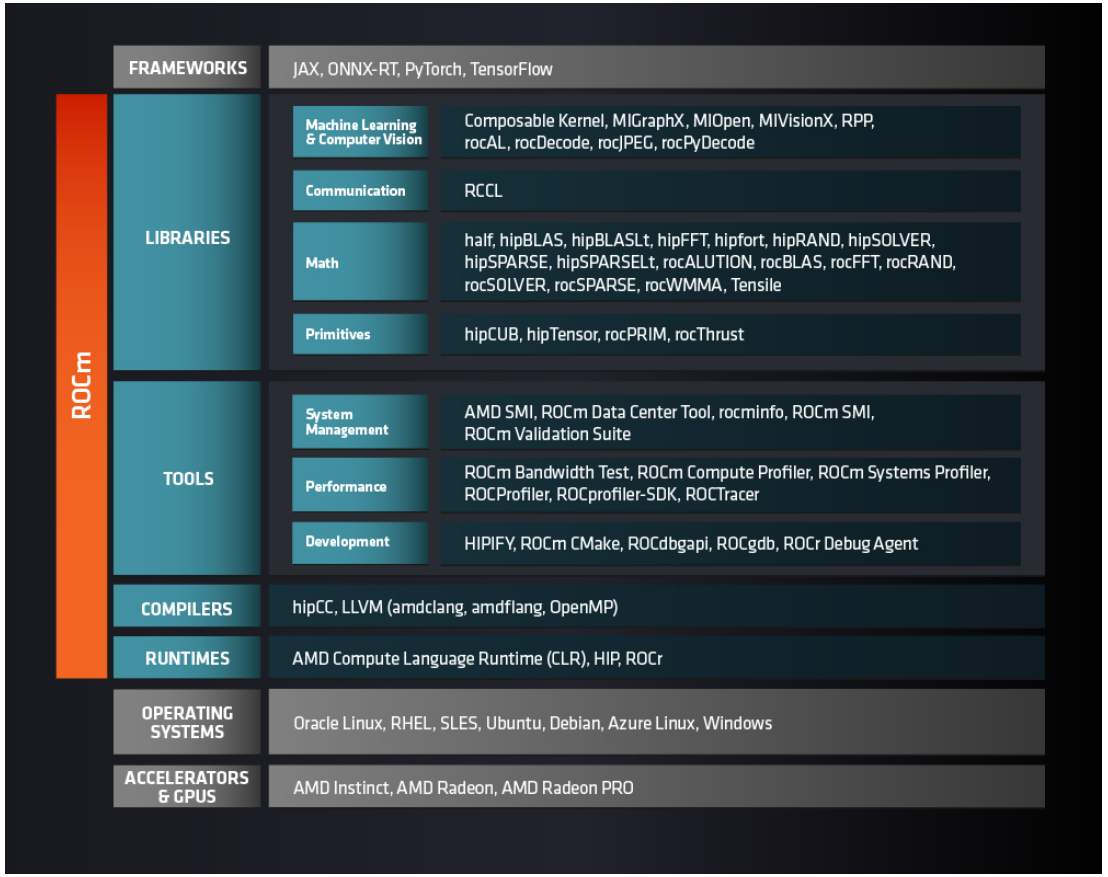


*Discrete GPU – Physical Device Memory; APU – Region of system for GPU direct access

From https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/instruction-set-architectures/amd-instinct-mi300-cdna3-instruction-set-architecture.pdf

AMD ROCm

- ROCm is AMD's open-source software stack for programming its GPUs
- Notable Supported languages and runtimes:
  - HIP (Heterogeneous-computing Interface for Portability)
  - OpenCL
  - OpenMP
  - SYCL
  - Hardware ISA
- AMDGPU LLVM backend support for compilation
- ROCm Runtime (ROCr/HSA) is the lowest-level of programming GPUs in ROCm
  - In charge of loading device binary onto GPUs
  - CUDA Driver's ROCm counterpart

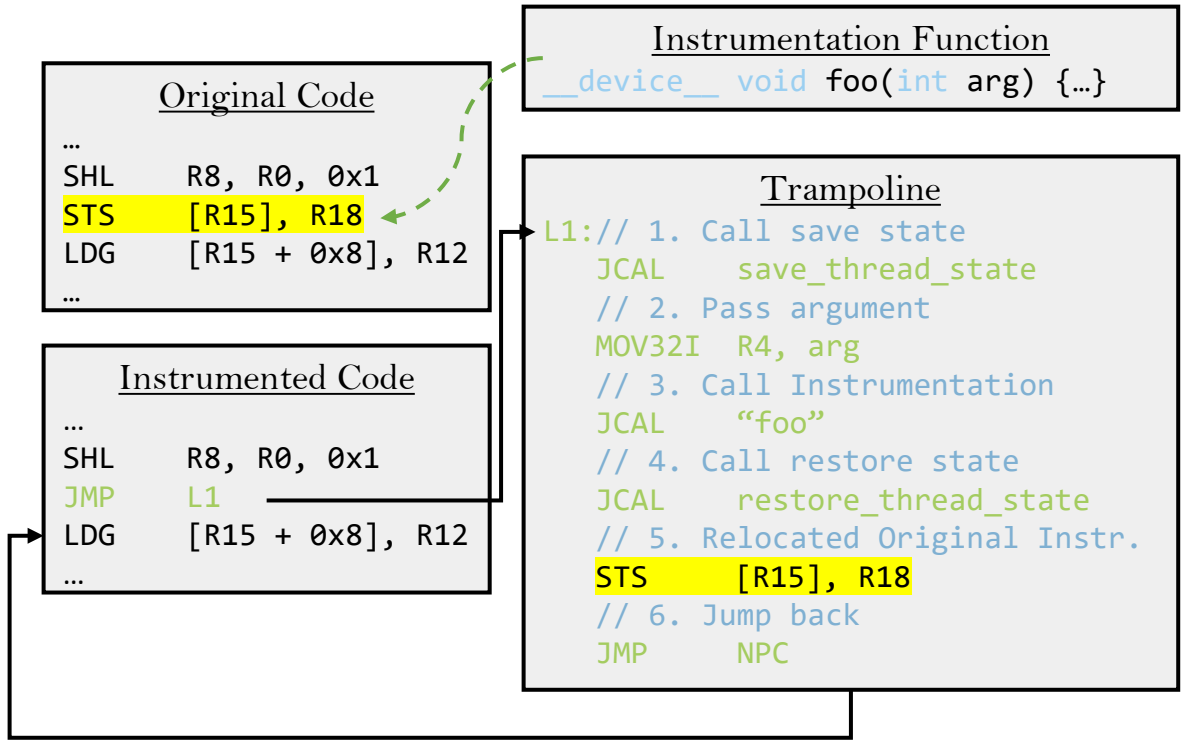


Taken from https://rocm.docs.amd.com/en/latest/_images/rocm-software-stack-6_3_2.jpg

Design Challenges

Challenge 1. Variable Length Instructions

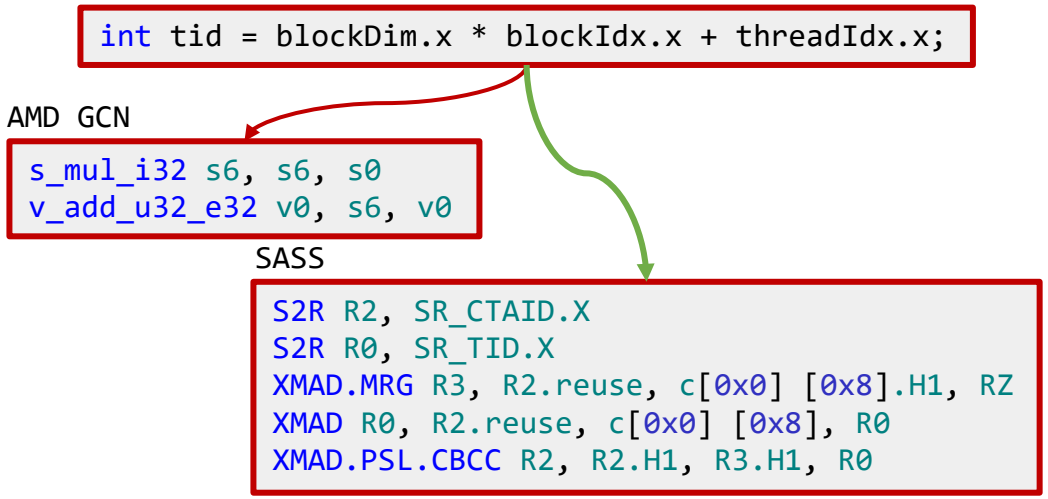
- ROCr does not allow fixed-address allocations of executable device memory regions
  - Cannot force jumps to trampolines to be short without a custom memory allocator
  - Makes instrumenting large kernels challenging
- AMD GPU long jumps require more than a single instruction to setup its target
  - More instructions need to be displaced
  - Two SGPRs need to be scavenged
  - Does not work if the instruction is close to the end of the kernel
- Renders instrumenting the "NVBit-way" near impossible without major modifications to ROCr



NVBit Instrumentation Overview
From "NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs", MICRO '19

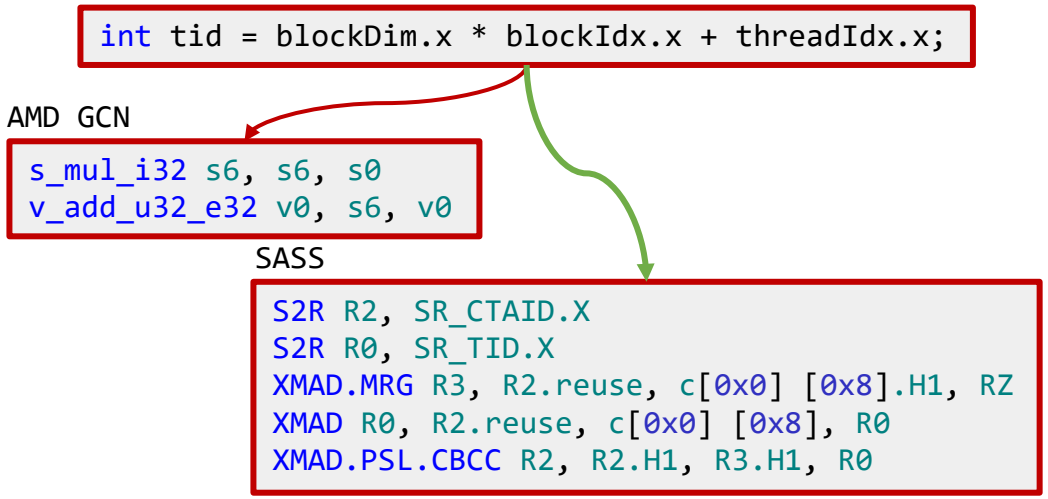
Challenge 2. Lowering Kernel Argument Intrinsics in Instrumentation Logic

- AMD GPUs have no read-only state registers unlike NVIDIA GPUs
  - They get passed to kernels inside SGPR as arguments instead
  - Get discarded right after the kernel is done using them
  - Notable values:
    - Queue Flat Scratch base
    - Wavefront scratch offset
    - Kernel argument address
    - Wavefront and work-item IDs



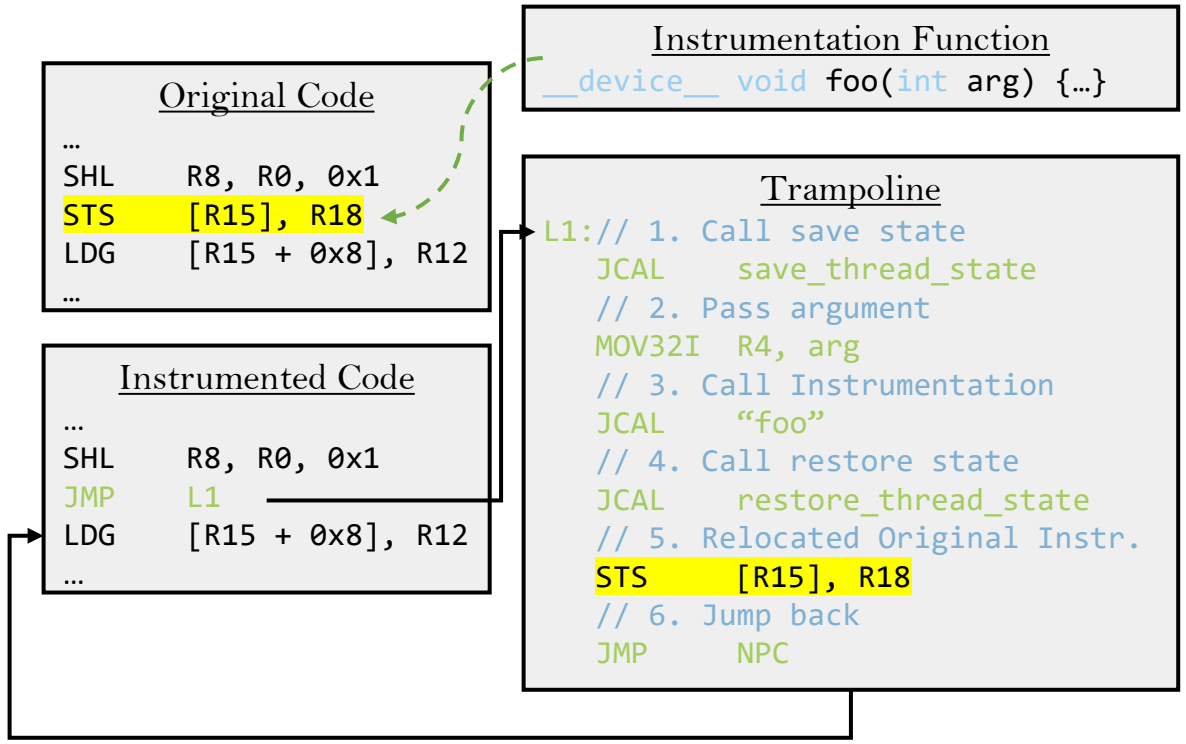
Challenge 2. Lowering Kernel Argument Intrinsics in Instrumentation Logic

- Requesting additional SGPR arguments might interfere with the original order of arguments
- Must emit additional instructions to:
  - Correct the argument order before the start of the kernel
  - Save the SGPR arguments needed by the instrumentation routines to be accessed later
- Must somehow use the saved SGPR arguments in the instrumentation functions



Challenge 3. No ROCr Loader Support for Device Functions

- ROCr does not have support for recognizing and dynamic linking device functions
  - Mainly due to potential kernel and device function register requirement mismatch
- Device functions get emitted in AMD GPU code, but they have a private binding
- We suspect both NVBit and GTPin take advantage of loader support for inserting calls to instrumentation functions
- Hard to link against instrumentation functions without loader support
  - Might not even work at the kernel driver level



NVBit Instrumentation Overview
From "NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs", MICRO '19

Challenge 4. Accessing Private Memory from Instrumentation Functions

- Instrumentation functions may need to spill registers to the stack without interfering with the target application state
- NVBit does not explicitly elaborate on how the instrumentation logic accesses the private segment
  - We suspect it primarily relies on the presence of a stack pointer register
- AMD GPUs have many different calling conventions across compute and graphics workload
  - Each with a different stack pointer
  - Assembly programs aren't required to follow calling conventions
- Must find a way to maintain an instrumentation stack without relying on calling conventions

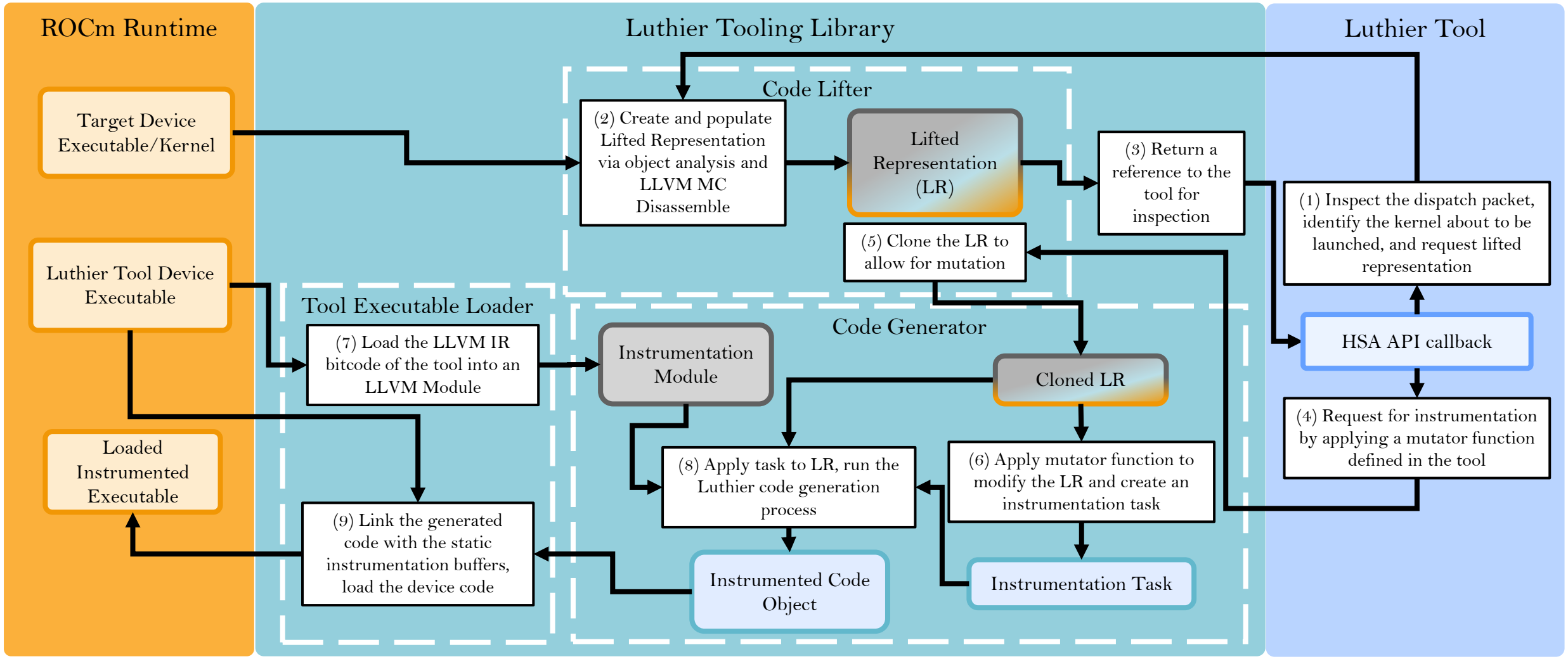
Table 29 AMDGPU Calling Conventions ¶

Calling Convention	Description
`ccc`	The C calling convention. Used by default. See Non-Kernel Functions for more details.
`fastcc`	The fast calling convention. Mostly the same as the `ccc`.
`coldcc`	The cold calling convention. Mostly the same as the `ccc`.
`amdgpu_cs`	Used for Mesa/AMDPAL compute shaders. ..TODO:: Describe.
`amdgpu_cs_chain`	Similar to `amdgpu_cs`, with differences described below. Functions with this calling convention cannot be called directly. They must instead be launched via the `llvm.amdgcn.cs.chain` intrinsic. Arguments are passed in SGPRs, starting at s0, if they have the `inreg` attribute, and in VGPRs otherwise, starting at v8. Using more SGPRs or VGPRs than available in the subtarget is not allowed. On subtargets that use a scratch buffer descriptor (as opposed to `scratch_{load,store}_*` instructions), the scratch buffer descriptor is passed in s[48:51]. This limits the SGPR / `inreg` arguments to the equivalent of 48 dwords; using more than that is not allowed. The return type must be void. Varargs, sret, byval, byref, inalloca, preallocated are not supported. Values in scalar registers as well as v0-v7 are not preserved. Values in VGPRs starting at v8 are not preserved for the active lanes, but must be saved by the callee for inactive lanes when using WWM (a notable exception is when the llvm.amdgcn.init.whole.wave intrinsic is used in the function - in this case the backend assumes that there are no inactive lanes upon entry; any inactive lanes that need to be preserved must be explicitly present in the IR). Wave scratch is "empty" at function boundaries. There is no stack pointer input or output value, but functions are free to use scratch starting from an initial stack pointer. Calls to `amdgpu_gfx` functions are allowed and behave like they do in `amdgpu_cs` functions. All counters (`lgkmcnt`, `vmcnt`, `storecnt`, etc.) are presumed in an unknown state at function entry. A function may have multiple exits (e.g. one chain exit and one plain `ret void` for when the wave ends), but all `llvm.amdgcn.cs.chain` exits must be in uniform control flow.
`amdgpu_cs_chain_preserve`	Same as `amdgpu_cs_chain`, but active lanes for VGPRs starting at v8 are preserved. Calls to `amdgpu_gfx` functions are not allowed, and any calls to `llvm.amdgcn.cs.chain` must not pass more VGPR arguments than the caller's VGPR function parameters.
`amdgpu_es`	Used for AMDPAL shader stage before geometry shader if geometry is in use. So either the domain (= tessellation evaluation) shader if tessellation is in use, or otherwise the vertex shader. ..TODO:: Describe.
`amdgpu_gfx`	Used for AMD graphics targets. Functions with this calling convention cannot be used as entry points. ..TODO:: Describe.
`amdgpu_gs`	Used for Mesa/AMDPAL geometry shaders. ..TODO:: Describe.

AMD GPU calling conventions, taken from https://llvm.org/docs/AMDGPUUsage.html#calling-conventions

Instrumentation Steps

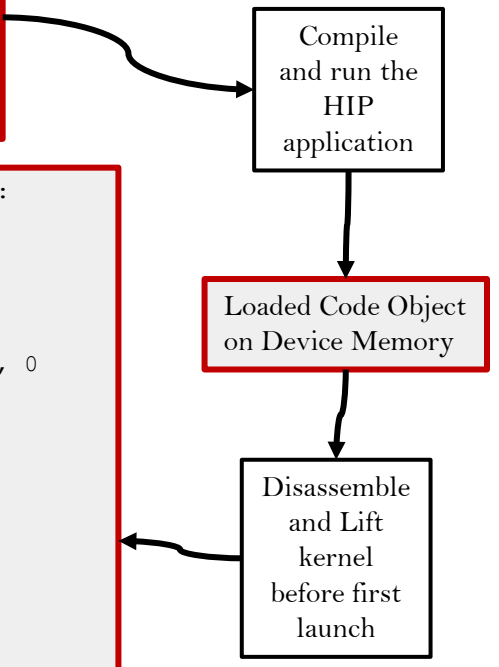
Luthier Instrumentation: At a Glance



Lifting Kernels to LLVM Machine IR

- Leverage the AMD GPU backend to disassemble kernel and device function loaded contents into LLVM MC instructions
- The disassembled contents and the ELF symbols of the inspected kernel will be combined and raised to LLVM Machine IR (MIR)
- Primary motivations:
  - Isolate kernel requirements inside a stand-alone ELF or Module for easy analysis with LLVM backend passes
  - Significantly more freedom instrumentation routines and move application instructions via LLVM APIs

```
// HIP Program Being Instrumented
__global__ void
vector_increment(int* __restrict__ a) {
 if (*a > 1)
    *a += 1;
}
```




```
# Machine code for function _Z16vector_incrementPi:
NoPHIs, TracksLiveness, NoVRegs, Selected
bb.0:
 successors: %bb.1(0x40000000), %bb.2(0x40000000);
%bb.1(50.00%), %bb.2(50.00%)

 $sgpr0_sgpr1 = S_LOAD_DWORDX2_IMM $sgpr4_sgpr5, 0, 0
 S_WAITCNT 49279
 $sgpr2 = S_LOAD_DWORD_IMM $sgpr0_sgpr1, 0, 0
 S_WAITCNT 49279
 S_CMP_LT_I32 $sgpr2, 2
 S_CBRANCH_SCC1 %bb.2
bb.1:
; predecessors: %bb.0
 successors: %bb.2(0x80000000); %bb.2(100.00%)

 $sgpr2 = S_ADD_I32 $sgpr2, 1
 $vgpr0 = V_MOV_B32_e32 0
 $vgpr1 = V_MOV_B32_e32 $sgpr2
 GLOBAL_STORE_DWORD_SADDR $vgpr0, $vgpr1, $sgpr0_sgpr1, 0
bb.2:
; predecessors: %bb.1, %bb.0

 S_ENDPGM 0
# End machine code for function _Z16vector_incrementPi.
```

Luthier Instrumentation: An Example

We want to count the number of times each scalar instruction is executed in the lifted representation obtained from the previous slide

We keep track of how the first instruction gets instrumented

```
# Machine code for function _Z16vector_incrementPi:
NoPHIs, TracksLiveness, NoVRegs, Selected
bb.0:
 successors: %bb.1(0x40000000), %bb.2(0x40000000);
%bb.1(50.00%), %bb.2(50.00%)

 $sgpr0_sgpr1 = S_LOAD_DWORDX2_IMM $sgpr4_sgpr5, 0, 0
 S_WAITCNT 49279
 $sgpr2 = S_LOAD_DWORD_IMM $sgpr0_sgpr1, 0, 0
 S_WAITCNT 49279
 S_CMP_LT_I32 $sgpr2, 2
 S_CBRANCH_SCC1 %bb.2
bb.1:
; predecessors: %bb.0
 successors: %bb.2(0x80000000); %bb.2(100.00%)

 $sgpr2 = S_ADD_I32 $sgpr2, 1
 $vgpr0 = V_MOV_B32_e32 0
 $vgpr1 = V_MOV_B32_e32 $sgpr2
 GLOBAL_STORE_DWORD_SADDR $vgpr0, $vgpr1, $sgpr0_sgpr1,
0
bb.2:
; predecessors: %bb.1, %bb.0

 S_ENDPGM 0
# End machine code for function _Z16vector_incrementPi.
```

Luthier Tool HIP Logic

Static device counter variable incremented before executing the intended instruction

Instrumentation function (hook) that will get called and inlined before the intended instruction

LLVM CodeGen APIs used to inspect the lifted kernel

Call the count instruction hook before each scalar instruction

Hooks can also take `llvm::MCRegister` and `llvm::Constant` values as arguments

```
__attribute__((device)) uint64_t Counter;

MARK_LUTHIER_DEVICE_MODULE

LUTHIER_HOOK_ANNOTATE countInstructionsScalar() {
(void)luthier::sAtomicAdd(&Counter, 1UL);
}

LUTHIER_EXPORT_HOOK_HANDLE(countInstructionsScalar);


static llvm::Error instrumentationLoop(
    luthier::InstrumentationTask &IT,
    luthier::LiftedRepresentation &LR) {
  for (auto &MBB : LR.getKernelMF()) {
    for (auto &MI : MBB) {
      bool IsScalar = luthier::isScalar(MI);
      bool IsLaneAccess = luthier::isLaneAccess(MI);
      if (IsScalar || IsLaneAccess)
        LUTHIER_RETURN_ON_ERROR(IT.insertHookBefore(
          MI,
LUTHIER_GET_HOOK_HANDLE(countInstructionsScalar)));
    }
  }
  return llvm::Error::success();
}
```

Luthier Intrinsics

A Luthier intrinsic binding to access the scalar atomic add instruction

```
__attribute__((device)) uint64_t Counter;

MARK_LUTHIER_DEVICE_MODULE

LUTHIER_HOOK_ANNOTATE countInstructionsScalar() {
(void)luthier::sAtomicAdd(&Counter, 1UL);
}

LUTHIER_EXPORT_HOOK_HANDLE(countInstructionsScalar);


static llvm::Error instrumentationLoop(
    luthier::InstrumentationTask &IT,
    luthier::LiftedRepresentation &LR) {
  for (auto &MBB : LR.getKernelMF()) {
    for (auto &MI : MBB) {
      bool IsScalar = luthier::isScalar(MI);
      bool IsLaneAccess = luthier::isLaneAccess(MI);
      if (IsScalar || IsLaneAccess)
        LUTHIER_RETURN_ON_ERROR(IT.insertHookBefore(
          MI,
LUTHIER_GET_HOOK_HANDLE(countInstructionsScalar)));
    }
  }
  return llvm::Error::success();
}
```

Luthier Intrinsics: Accessing Low-level Instructions in Instrumentation Hooks

- **Inline Assembly Problems:**
  - Inline Assembly physical register usage is not known until the very last step of compilation in LLVM
    - Luthier requires knowledge of register liveness before each instruction
  - Makes unnecessary copies of the same register
    - Does not help with scarcity of registers during instrumentation
- Luthier intrinsic is provided as a better alternative to inline assembly
- Luthier's code generator will lower these intrinsics to a sequence of machine instructions in SSA form
- Luthier already has a set of pre-defined intrinsics
  - Can be extended by the tool writer

Inst. Module IR At the Start of the Luthier Code Generation Process

Counter Variable

Its definition is removed in the tool compilation process offline, since a copy has already been loaded by ROCr

Count instruction hook LLVM IR

Intrinsic binding declaration

Does not have a body similar to LLVM intrinsics

```
target triple = "amdgcn-amd-amdhsa"

@Counter = external addrspace(1) externally_initialized
global i64, align 8

; Function Attrs: alwaysinline convergent mustprogress
nounwind
define internal void @countInstructionsScalar() #5 {
 %1 = tail call noundef i64
@"luthier::sAtomicAdd.i64.ptr.i64"(ptr noundef
addrspacecast (ptr addrspace(1) @Counter to ptr), i64
noundef 1) #7
 ret void
}

; Function Attrs: convergent mustprogress noinline nounwind
declare dso_local noundef i64
@"luthier::sAtomicAdd.i64.ptr.i64"(ptr noundef, i64
noundef) #6
```

Inst. Module IR: After Generating a Call to the Instrumentation Hook

An "Injected Payload" function for the first instruction in the kernel

It takes no arguments and returns nothing

Contains a sequence of call instructions to instrumentation hooks

The machine instructions of this function will be patched in before the target instruction

```
target triple = "amdgcn-amd-amdhsa"

@Counter = external addrspace(1) externally_initialized
global i64, align 8

; Function Attrs: alwaysinline convergent mustprogress
nounwind
define internal void @countInstructionsScalar() #5 {
 %1 = tail call noundef i64
@"luthier::sAtomicAdd.i64.ptr.i64"(ptr noundef
addrspacecast (ptr addrspace(1) @Counter to ptr), i64
noundef 1) #8
 ret void
}

; Function Attrs: convergent mustprogress noinline nounwind
declare dso_local noundef i64
@"luthier::sAtomicAdd.i64.ptr.i64"(ptr noundef, i64
noundef) #6

; Function Attrs: naked
define void @"MI: $sgpr0_sgpr1 = S_LOAD_DWORDX2_IMM
$sgpr4_sgpr5, 0, 0, MMB ID: 0"() #7 {
 call void @countInstructionsScalar()
 ret void
}
```

Inst. Module IR: After IR Optimization Passes

The call to the instrumentation hook is now inlined inside the "Injected Payload" function

The instrumentation hook has been dead-code eliminated

```
target triple = "amdgcn-amd-amdhsa"

@Counter = external addrspace(1) externally_initialized
global i64, align 8

; Function Attrs: convergent mustprogress noinline nounwind
declare dso_local noundef i64
@"luthier::sAtomicAdd.i64.ptr.i64"(ptr noundef, i64
noundef) local_unnamed_addr #0

; Function Attrs: naked
define void @"MI: $sgpr0_sgpr1 = S_LOAD_DWORDX2_IMM
$sgpr4_sgpr5, 0, 0, MMB ID: 0"() #1 {
 %1 = tail call noundef i64
@"luthier::sAtomicAdd.i64.ptr.i64"(ptr noundef
addrspacecast (ptr addrspace(1) @Counter to ptr), i64
noundef 1) #2
 ret void
}
```

Inst. Module MIR: Before Register Allocation

VGPR2 is the "State Value Array" load location

It will remain there until the end of the injected payload

Encode the register liveness requirements of the target instruction in valid LLVM MIR syntax

This allows the the register allocator to spill the live registers however it sees fit, if needed

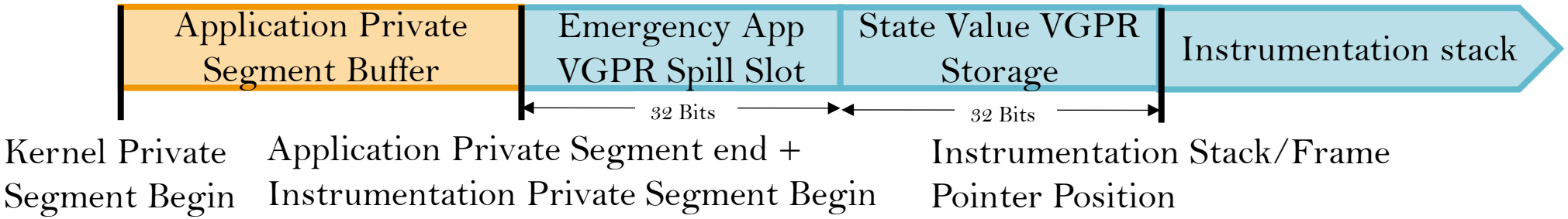
```
# Machine code for function MI: $sgpr0_sgpr1 =
S_LOAD_DWORDX2_IMM $sgpr4_sgpr5, 0, 0, MMB ID: 0: IsSSA,
TracksLiveness

bb.0 (%ir-block.0):
 liveins: $vgpr2, $sgpr4, $sgpr5
 %6:sreg_32 = COPY killed $sgpr4
 %7:sreg_32 = COPY killed $sgpr5
 %3:sreg_64 = SI_PC_ADD_REL_OFFSET target-flags(amdgpu-
gotprel32-lo) @Counter, target-flags(amdgpu-gotprel32-hi)
@Counter, implicit-def dead $scc
 %4:sreg_64_xexec = S_LOAD_DWORDX2_IMM killed %3:sreg_64,
0, 0 :: (dereferenceable invariant load (s64) from got,
addrspace 4)
 %5:sreg_64 = S_MOV_B64 1
 %1:sgpr_64 = COPY %4:sreg_64_xexec
 %2:sgpr_64 = COPY %5:sreg_64
 %0:sgpr_64 = S_ATOMIC_ADD_X2_IMM_RTN %2:sgpr_64(tied-def
0), %1:sgpr_64, 0, 0
 $sgpr4 = COPY killed %6:sreg_32
 $sgpr5 = COPY killed %7:sreg_32
 SI_RETURN implicit $vgpr2, implicit $sgpr4, implicit
$sgpr5

# End machine code for function MI: $sgpr0_sgpr1 =
S_LOAD_DWORDX2_IMM $sgpr4_sgpr5, 0, 0, MMB ID: 0.
```

State Value Array

- State value array is a fixed array of 64 32-bit values that can be stored in a single VGPR of a single wavefront
  - Also referred to as **Whole-Wave Mode (WWM)** register in the LLVM AMDGPU backend
- The state value array can be used to:
  - Spill slots for setting up/tearing down instrumentation stack frame
  - Store values shared among threads in a wave (e.g., kernel SGPR arguments, private segment buffer, start of instrumentation stack, etc.)
- All injected payloads can access the state value array using Luthier intrinsics



State Value Array: Storage

- The state value array can be stored in unused registers of the target application:
  - On a single unused VGPR, or a Single SGPR on newer targets to hold the instrumentation stack pointer
- If no fixed storage is found, the state value is moved between these storage schemes
  - Additional code must be inserted between app instructions to move the state value between storage schemes during the patching step
  - Similar to Hsuan-Heng Wu's scheme presented in the past tools workshop

Inst. Module MIR: After LLVM's Prologue/Epilogue Insertion Pass

After this pass, we check if an injected payload requires access to the state value array, and if so:

- Emits code to load/store the state value register to/from its storage
- Generate code at the end of the code generation pipeline for initializing the state value register and storing it for later use

Since the state value array is not used here, the code does not change

```
# Machine code for function MI: $sgpr0_sgpr1 =
S_LOAD_DWORDX2_IMM $sgpr4_sgpr5, 0, 0, MMB ID: 0: NoPHIs,
TracksLiveness, NoVRegs, TiedOpsRewritten,
TracksDebugUserValues

bb.0 (%ir-block.0):
 liveins: $sgpr4, $sgpr5, $vgpr2
 renamable $sgpr6_sgpr7 = SI_PC_ADD_REL_OFFSET target-
flags(amdgpu-gotprel32-lo) @Counter, target-flags(amdgpu-
gotprel32-hi) @Counter, implicit-def dead $scc
 renamable $sgpr6_sgpr7 = S_LOAD_DWORDX2_IMM killed
renamable $sgpr6_sgpr7, 0, 0 :: (dereferenceable invariant
load (s64) from got, addrspace 4)
 renamable $sgpr8_sgpr9 = S_MOV_B64_IMM_PSEUDO 1
 dead renamable $sgpr8_sgpr9 = S_ATOMIC_ADD_X2_IMM_RTN
killed renamable $sgpr8_sgpr9(tied-def 0), killed renamable
$sgpr6_sgpr7, 0, 0
 SI_RETURN implicit $vgpr2, implicit $sgpr4, implicit
$sgpr5

# End machine code for function MI: $sgpr0_sgpr1 =
S_LOAD_DWORDX2_IMM $sgpr4_sgpr5, 0, 0, MMB ID: 0.
```

Inst. Module MIR: Final Form

```
# Machine code for function MI: $sgpr0_sgpr1 = S_LOAD_DWORDX2_IMM $sgpr4_sgpr5, 0, 0, MMB ID: 0: NoPHIs,
TracksLiveness, NoVRegs, TiedOpsRewritten, TracksDebugUserValues

bb.0 (%ir-block.0):
 liveins: $sgpr4, $sgpr5, $vgpr2
 S_WAITCNT 0
 BUNDLE implicit-def $sgpr6_sgpr7, implicit-def $sgpr6, implicit-def $sgpr6_lo16, implicit-def $sgpr6_hi16,
implicit-def $sgpr7, implicit-def $sgpr7_lo16, implicit-def $sgpr7_hi16, implicit-def $scc {
   $sgpr6_sgpr7 = S_GETPC_B64
   $sgpr6 = S_ADD_U32 internal $sgpr6, target-flags(amdgpu-gotprel32-lo) @Counter + 4, implicit-def $scc
   $sgpr7 = S_ADDC_U32 internal $sgpr7, target-flags(amdgpu-gotprel32-hi) @Counter + 12, implicit-def $scc,
implicit internal $scc
 }
 renamable $sgpr6_sgpr7 = S_LOAD_DWORDX2_IMM killed renamable $sgpr6_sgpr7, 0, 0 :: (dereferenceable invariant
load (s64) from got, addrspace 4)
 renamable $sgpr8_sgpr9 = S_MOV_B64 1
 S_WAITCNT 49279
 dead renamable $sgpr8_sgpr9 = S_ATOMIC_ADD_X2_IMM_RTN killed renamable $sgpr8_sgpr9(tied-def 0), killed
renamable $sgpr6_sgpr7, 0, 0
 S_WAITCNT 49279
 BUFFER_WBINVL1_VOL implicit $exec
 S_SETPC_B64_return undef $sgpr30_sgpr31, implicit killed $vgpr2, implicit killed $sgpr4, implicit killed $sgpr5

# End machine code for function MI: $sgpr0_sgpr1 = S_LOAD_DWORDX2_IMM $sgpr4_sgpr5, 0, 0, MMB ID: 0
```

Final Step: Patching the Lifted Representation

- After the injected payload MIR has been generated, it will be patched into the Lifted Representation to obtain the final instrumented kernel:
  - Injected payload's contents will be patched before their respective instructions
  - Any used Global Value will also be patched in
  - If present, stack operands will be patched on top of the original kernel's stack
  - If the state value array has been used, code will be emitted to set it up at the beginning of the kernel

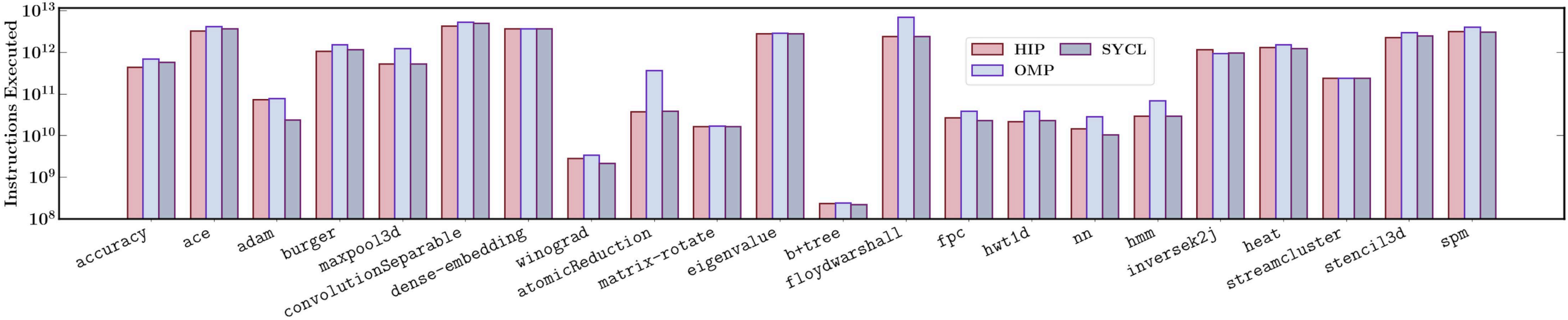
Final Instrumented Kernel

```
        .amdgcn_target "amdgcn-amd-amdhsa-unknown-
gfx908"
        .amdhsa_code_object_version 6
        .text
        .protected      _Z16vector_incrementPi
        .weak   _Z16vector_incrementPi
        .p2align        8
        .type   _Z16vector_incrementPi,@function
_Z16vector_incrementPi:
        s_branch .LBB0_4
.LBB0_0:
        s_load_dwordx2 s[0:1], s[4:5], 0x0
        s_waitcnt lgkmcnt(0)
        s_load_dword s2, s[0:1], 0x0
        s_waitcnt lgkmcnt(0)
        s_cmp_lt_i32 s2, 2
        s_cbranch_scc1 .LBB0_2
        s_add_i32 s2, s2, 1
        v_mov_b32_e32 v0, 0
        v_mov_b32_e32 v1, s2
        global_store_dword v0, v1, s[0:1]
.LBB0_2:
        s_endpgm
```

```
.LBB0_4:
        s_waitcnt vmcnt(0) expcnt(0) lgkmcnt(0)
        s_getpc_b64 s[6:7]
        s_add_u32 s6, s6, Counter@gotpcrel32@lo+4
        s_addc_u32 s7, s7, Counter@gotpcrel32@hi+12
        s_load_dwordx2 s[6:7], s[6:7], 0x0
        s_mov_b64 s[8:9], 1
        s_waitcnt lgkmcnt(0)
        s_atomic_add_x2 s[8:9], s[6:7], 0x0
        s_waitcnt lgkmcnt(0)
        buffer_wbinvl1_vol
        s_branch .LBB0_0
```

Use Case

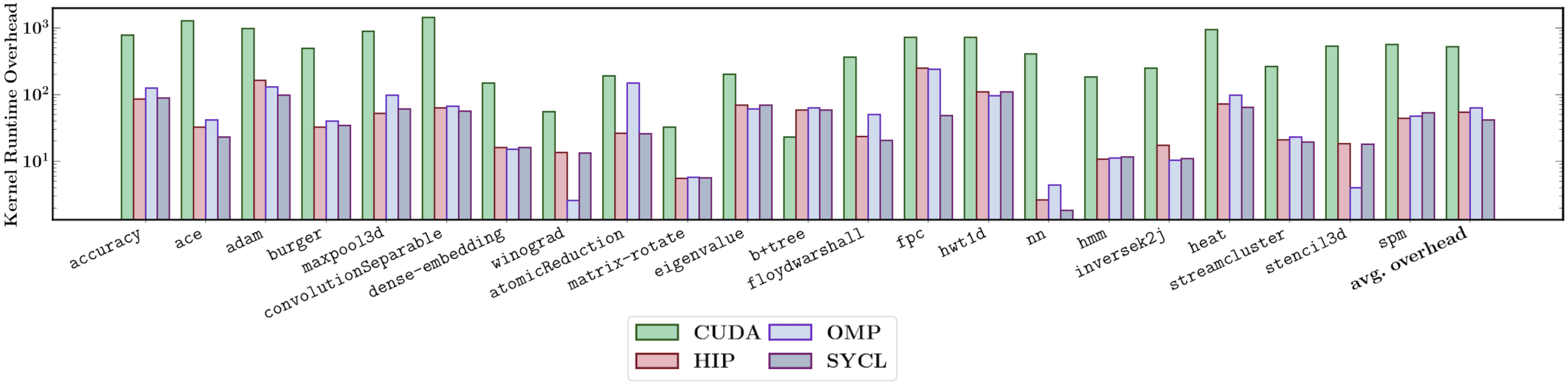
Instructions Executed in the HeC Benchmark Suite*



HIP, OMP and SYCL Workloads

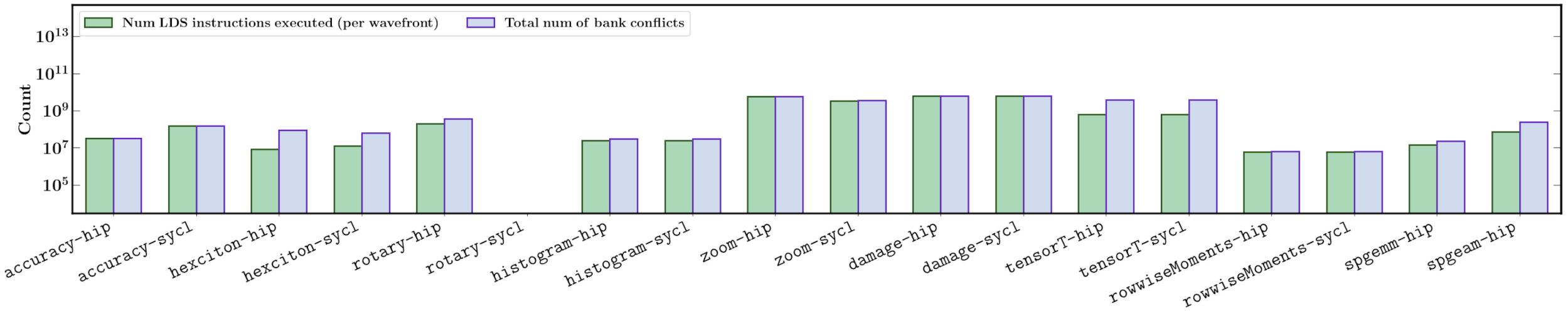
*Z. Jin and J. S. Vetter, "A Benchmark Suite for Improving Performance Portability of the SYCL Programming Model," *2023* IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Raleigh, NC, USA, *2023*, pp. *325-327*, doi: 10.1109/ISPASS57527.2023.00041. (https://ieeexplore.ieee.org/document/10158214)

Instructions Executed in the HeC Benchmark Suite



- Luthier incurs a ~50X kernel runtime overhead on average for instruction counting
- NVBit has a ~500X runtime overhead on the equivalent CUDA benchmarks!

Local Data Share Bank Conflicts in the HeC Benchmark Suite (HIP and SYCL)



- AMD GPUs leverage Local Data Share (LDS) to improve performance by providing a shared memory space within a workgroup
- Allows different threads within a workgroup to efficiently access the same data with lower latency compared to global memory

Conclusion and Future Work

Conclusion

- We presented our DBI framework for AMD GPUs
  - We presented challenges faced when instrumenting AMD GPU binaries
  - We showed how Luthier's design effectively addresses each challenge
- Luthier's instrumented code generation scheme can also be adapted to work with other GPU instrumentation frameworks (if LLVM backend support is present)
  - Can also be considered in use with custom accelerators

Current Focus

- Support cascading Luthier/rocprofiler-sdk tools
  - Instrumenting an already-instrumented application
  - Profiling an instrumented kernel
- More efficient tracing via API tables instead of using C-style callbacks
- Pass dynamically-allocated instrumentation buffers to kernels as kernel arguments
  - Relax requirement for serializing kernel launches
- Support instrumenting kernels calling function pointers
- Support patching instrumentation logic into large kernels
- Test Luthier on additional AMD GPU targets

Acknowledgements

- AMD*
  - Timour Paltashev
  - Mathew Arsenault
  - Jacob Lambert
  - Konstantin Zhuravlyov
  - Shilei Tian
  - Tony Tye
  - Rene Van Oostrum
  - Benjamin Welton
  - Sadik Armagan
- NUCAR Undergraduate Students
  - Andrew Nguyen
  - Omar Shair
  - Maya De Los Santos
  - Ivan Rosales
  - Ruben Noroian
  - Danielle Mclaughlin
- NUCAR Masters Student
  - Qishi Wang
- NUCAR PhD Students
  - Yuhui Bao
  - Zlatan Feric
  - Aymane El Jerari
- You can read our **IEEE ISPASS 2025 paper!**

***We thank AMD for their generous support of this project**

Thank You